



H.E.B. ECOLE SUPERIEUR D'INFORMATIQUE

LABORATOIRE DE C++ : PROJET 2

Starlight

Auteurs :
Paul KRIWIN
Simon PLACENTINO

Titulaire du cours :
Dr. Romain ABSIL

21 avril 2015

Table des matières

1	Introduction	2
2	Conventions	3
2.1	Langage et compilation	3
2.2	Documentation	4
2.3	Contenu des fichiers	4
2.4	Fichiers	5
2.4.1	noms	5
2.4.2	headers	5
2.4.3	sources	5
2.4.4	test	5
2.5	Nom des namespace	5
2.6	Nom des classes	5
2.7	Nom des variables	6
3	Les classes	7
3.1	Les objets géométriques	7
3.1.1	Ellipse	7
3.1.2	Droite	8
3.1.3	Rectangle	9
3.1.4	Point	9
3.1.5	Utilitaire	10
3.2	Les éléments	11
3.2.1	Element	11
3.2.2	Cristal	11
3.2.3	Destination	12
3.2.4	Lentille	12
3.2.5	Miroir	12
3.2.6	Bombe	13
3.2.7	Rayon	13
3.2.8	Source	13
3.2.9	Mur	13
3.2.10	Niveau	14

3.2.11	Createur de niveau	14
3.3	L'exception	14
3.3.1	Exception Starlight	14
3.4	Les objets visuels	15
4	Structure du programme	16
5	Algorithmes	17
5.1	Réflexion	17
5.2	Intersection	18
5.2.1	Deux droites	18
5.2.2	Droite et rectangle	18
5.2.3	Droite et ellipse	18
6	Test effectués	19
6.1	Framework de test	19
6.2	Tests unitaire	19
7	Conclusion	20
A	Références	21

Chapitre 1

Introduction

Chapitre 2

Quelques conventions utilisées

2.1 Langage et compilation

Ce projet a été bâti à l'aide de **Qt Creator 5.4**¹, pour l'accès à la bibliothèque graphique, et compilé à l'aide de **g++**² dans sa version 4.8 (et compatible 4.9). La norme **ISO/IEC 14882 :2011**³, aussi appelée **c++11**, est celle utilisée. A cela s'ajoute certains flags⁴ de compilation

`g++ -std=c++11 -Wextra -Wall -pedantic-errors`

- **-std=c++11** pour travailler en **c++11**,
- **-Wextra** qui nous permet d'avoir des messages d'avertissements,
- **-Wall** qui nous permet d'avoir tous les messages d'avertissements,
- **-pedantic-errors** qui transforme certains warnings⁵ en erreurs.

Pour des raisons pratiques, le projet est accompagné d'un fichier de type **Makefile**⁶ qui s'occupe de

- nettoyer le répertoire à l'aide de la commande
`make_clean`
- compiler l'ensemble du projet à l'aide de la commande
`make`

1. <http://www.qt.io/developers/>

2. <https://gcc.gnu.org/>

3. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

4. un flag est un argument de compilation optionnel

5. un warning n'empêche pas une bonne compilation mais demande au programmeur de vérifier

6. https://www.gnu.org/software/make/manual/html_node/Makefiles.html

2.2 Documentation

L'ensemble du projet est documenté à l'aide des balises Qt

```
/*!  
 * \brief Documentation de la methode enTete  
 * \param Description d'un parametre  
 * \return Ce qui est retourne  
 * \see Renvoie vers une autre documentation  
 */  
T enTete(R param);  
  
/*!  
 * \brief Documentation de la variable  
 */  
T variable;
```

et cette documentation est uniquement présente dans les fichiers **headers**. La documentation, au format L^AT_EXet html, a été générée à l'aide de l'outil Doxygen⁷ dans sa dernière version disponible⁸. Les spécifications de compilations sont disponible dans le document “doxyConfig”. La documentation U.M.L.⁹ a été générée par l'intégration de l'outil **graphviz**¹⁰. Cette documentation est disponible dans les dossiers

documentation/html/index.html
documentation/latex/documentation.pdf

2.3 Contenu des fichiers

Chaque fichier contiendra les en-têtes, ou le code source, d'au maximum une seule classe et d'au maximum un seul namespace. Si la classe ne contient pas au moins une classe publique, le fichier portera le nom du namespace. Par exemple, le contenu du fichier hello.hpp peut être ainsi :

```
namespace unnamespace {class Hello {};
```

```
namespace hello {}
```

```
class Hello {};
```

Le contenu de l'ensemble des fichiers est disponible dans le dossier

src/

7. <http://doxygen.org/>

8. 1.8.9.1

9. <http://www.uml.org/>

10. disponible dans les packages GNU/Linux - Ubuntu standards

2.4 Fichiers

2.4.1 noms

Le nom des fichiers est entièrement écrit en minuscule et porte le nom de la classe, ou du namespace qu'il contient.

2.4.2 headers

Les headers porteront l'extension **hpp**. Ils seront disposés dans le même dossier que leur fichier source **cpp** correspondant.

2.4.3 sources

Les fichiers sources porteront l'extension **cpp**. Ils seront disposé dans le même dossier que leur fichier header **hpp** correspondant.

2.4.4 test

Le nom des fichiers de test est composé du nom de la classe ou du namespace testé suivi de "test"

```
nuke.cpp → nuketest.cpp  
mirror.cpp → mirrortest.cpp
```

L'ensemble de ces classes sont disponibles dans le dossier

`test/`

2.5 Nom des namespace

Le nom d'un namespace sera exclusivement en minuscule.

```
namespace hebesi {}
```

2.6 Nom des classes

Les classes commencent toutes par une majuscule pour continuer, ensuite, en CamelCase

```
class UneBonneClasse {};  
class uneMauvaiseclasse {};
```


2.7 Nom des variables

Toutes les variables sont écrites en **camelCase**¹¹

```
T uneBonneVariable ;  
T UneMauvaiseVariable ;  
T uneautreMauvaisevariable ;
```

et possède des noms le plus explicite possible

```
T nb; // OK  
T n{38.}; // NOK  
T waveLength; // OK
```

Les variables de classes possèdent le même nom que le paramètre de constructeur qui l'initialisera. Le langage nous donne la possibilité de désambiguïser l'utilisation de ces variables à l'aide de

```
T var ;  
this -> var ;
```

et de la liste d'initialisation

```
UneClasse :: UneClasse (T param) : param {param} {}
```

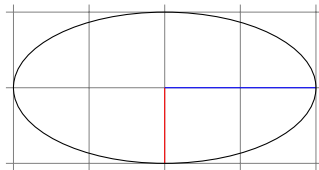
11. <https://fr.wikipedia.org/wiki/CamelCase>

Chapitre 3

Présentation succincte des classes

3.1 geometry

3.1.1 ellipse.hpp



Une ellipse est un objet géométrique à deux dimensions représentée par une courbe plane fermée obtenu par découpe d'un cône sur un plan. Si ce dernier est perpendiculaire à l'axe du cône, l'ellipse sera alors un cercle. Éléments caractéristiques d'une ellipse :

- une coordonnée cartésienne, ou polaire, de son **centre** c ,
- une distance séparant le centre de l'intersection avec la tangente parallèle à l'axe des ordonnées x_{radius} ,
- une distance séparant le centre de l'intersection avec la tangente parallèle à l'axe des abscisses y_{radius} .

Ces éléments nous permettront de tracer une ellipse selon l'équation :

$$\frac{(x - c_x)^2}{x_{radius}^2} + \frac{(y - c_y)^2}{y_{radius}^2} = 1$$

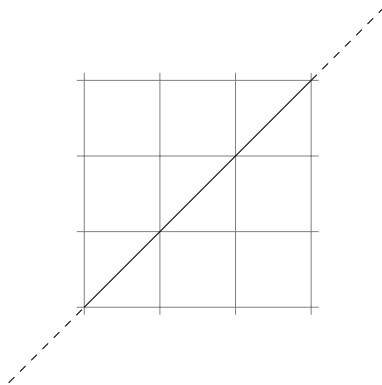
Cette classe peut tout à fait être instancié en objet géométrique elliptique et possède des méthodes d'interactions avec une droite¹.

```
Point * Ellipse::getIntersectionsPoints(const Line &);
```

1. L'algorithme est disponible dans la section du même nom ou dans la documentation accompagnant le projet

Dans le contexte présent, certains éléments du jeu seront des ellipse par le phénomène d’héritage mis en place dans le paradigme orienté objet de C++²

3.1.2 line.hpp



Une droite est une ligne sans épaisseur, rectiligne et infinie dans le plan. Pour exister, une droite aura besoin :

- d’un coefficient angulaire $m = \frac{\Delta y}{\Delta x}$ représentant la distance à parcourir sur l’axe des ordonnées pour une unité de distance sur l’axe des abscisses.
- d’un terme indépendant $p = \frac{y}{m \cdot x}$ représentant le décalage de chaque point sur l’axe des ordonnées,
- ou de deux points de coordonnées dans le plan a, b
- ou d’un point de coordonnées dans le plan a et d’un coefficient angulaire $m = \frac{\Delta y}{\Delta x}$.

Ces éléments nous permettent de tracer une droite selon l’équation :

$$y = m \cdot x + p$$

Il s’agit donc de cette dernière qui a été modélisé. Pour la gestion des droites verticales, un paramètre supplémentaire, et optionnel, a été rajouté au constructeur. Ainsi, une méthode de la classe utilities nous permet de savoir si le coefficient angulaire est dit “infini” et donc nous permettre de contenter la demande

```
bool * Line::isVertical();
```

Ainsi, il est simple de savoir, en interieur comme en exterieur de la classe, si la droite est verticale ou non. Il est aussi possible de savoir si un point fait parti de la droite courante.

```
bool * Line::includes(const Point &) const;
```

2. http://www.tutorialspoint.com/cplusplus/cpp_object_oriented.htm

Cette classe peut tout à fait être instancié en objet géométrique linéaire et possède des méthodes d'interactions avec d'autres droites³.

```
Point * Line::getIntersectionPoint(const Line &);
```

Dans le contexte présent, certains éléments du jeu seront des droites par le phénomène d'héritage mis en place dans le paradigme orienté objet de C++⁴

3.1.3 rectangle.hpp



Un rectangle est une forme géométrique à 4 segments de droite⁵ parallèle deux à deux. Ceux-ci vont donc former 4 angles droit ($\frac{\pi}{2}rad$) Cette forme peut être représentée par :

- la coordonnée du coin supérieur gauche $Sg = (x, y)$
- la grandeur des deux segments formant un angle de $\frac{\pi}{2}rad$ en ce point *hauteur* et *largeur*.

Ainsi, il sera aisé de déterminer la position des autres coins

- $Sd = (Sg_x + largeur, Sg_y)$
- $Ig = (Sg_x, Sg_y + hauteur)$
- $Id = (Sg_x + largeur, Sg_y + hauteur)$

et de modéliser le rectangle à l'aide de 4 équations de droite, les objets Line. Cette classe peut tout à fait être instancié en objet géométrique rectangulaire et possède des méthodes d'interactions avec d'autres droites⁶.

```
vector<Point> Rectangle::getIntersectionPoints(Line &);
```

3.1.4 point.hpp

Un point est un objet mathématique permettant de situer un élément dans un plan ou dans l'espace. Dans notre cas, plus spécifiquement dans un plan à deux dimensions. Celui-ci peut-être représenté de plusieurs manières dans le plan cartésien⁷ :

- sous la forme d'une coordonnées cartésienne à l'aide de
- une origine,

3. L'algorithme est disponible dans la section du même nom ou dans la documentation accompagnant le projet

4. http://www.tutorialspoint.com/cplusplus/cpp_object_oriented.htm

5. Un segment de droite est une partie de droite délimitée par deux points non confondus

6. L'algorithme est disponible dans la section du même nom ou dans la documentation accompagnant le projet

7. <http://www.cslaval.qc.ca/sitsatl11/maths2003/cartesien.html>

- deux vecteurs partant de cette origine et perpendiculaires,

$$P = (\vec{x}, \vec{y})$$

- et sous la forme d’une coordonnée polaire à l’aide de
 - une origine,
 - une coordonnée radiale r ,
 - une coordonnée angulaire α .

3.1.5 utilities.hpp

Le namespace “utilities” mis en place ici est un ensemble de fonctions et valeurs constantes spécifiquement définies pour les calculs intervenant dans le projet.

constantes :

PI est une approximation de π sur 26 décimales,

PI_2 est une approximation de $\frac{\pi}{2}$ sur 26 décimales,

PI_4 est une approximation de $\frac{\pi}{4}$ sur 26 décimales,

EPSILON est une marge d’erreur de 10^{-7} ,

INF représente un nombre dit “infini” dans le milieu informatique.

fonctions :

Resolution d’équation du second degre

```
utilitaire::secondDegreeEquationSolver
```

Transforme un angle exprime en radian en un angle exprime en degres

```
utilitaire::angleAsDegree
```

Permet de tester l’egalite ou l’inegalite entre deux nombre reels a un Epsilon d’erreur

```
utilitaire::equals
utilitaire::greaterOrEquals
utilitaire::lessOrEquals
```

Permet de trouver le coefficient angulaire a partir de deux points

```
utilitaire::slopeFromPoints
```

Permet de trouver la valeur tangente d'un angle en radian mais aussi de retourner une valeur particuliere pour la tangente de $\pi/2$

```
utilitaire::tan
```

Permet de savoir si α vaut $\frac{\pi}{2} + n * \pi$ $n \in \mathbb{N}$

```
utilitaire::isHalfPiPlusNPi
```

Pour d'autres informations, consultez la documentation générée.

3.2 éléments

3.2.1 element.hpp

Cette classe, abstraite et donc non instanciable, représente un élément du jeu lié à un, et un seul, niveau du jeu. Cette classe permet donc d'établir une communication entre les différents éléments du niveau et le niveau lui-même à travers un référencement de ce dernier. Par le phénomène d'héritage, chaque élément se devra d'établir sa manière propre de réagir avec le niveau en lui exprimant :

- ses points d'intersection avec un rayon si il lui sont demandés,

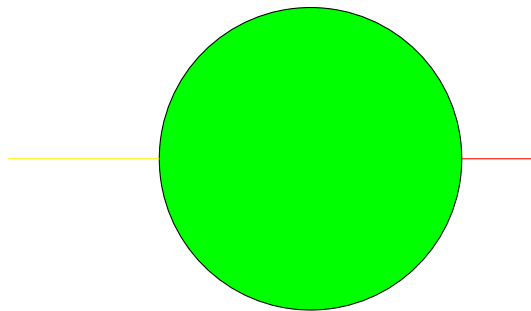
```
Element::includeRay(Ray);
```

- les actions à effectuer si un contact avec le rayon a eu lieu.

```
Element::reactToRay(Ray);
```

Il sera donc aisé pour le niveau de gérer les collisions des éléments de manière anonyme et optimale (cf Niveau).

3.2.2 crystal.hpp



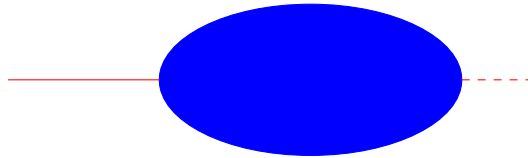
Cette classe est, par héritage, un élément ainsi qu'une ellipse. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. Le cristal modifie la longueur d'onde d'un rayon qui le traverse. D'un point de vue technique, il en relève

donc d'un choix personnel, il ne va pas amplifier le rayon entrant à partir de sa sortie mais arrêtera ce rayon et en créera un nouveau qui, lui, aura subi l'action du cristal.

3.2.3 dest.hpp

Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La destination est le but du jeu et l'atteindre terminera automatiquement la partie sur une victoire.

3.2.4 lens.hpp



Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La lentille est un éventuel obstacle à un rayon puisqu'il ne le laissera passer que si la longueur d'onde de ce dernier respecte le critère

$$lens_{min} \leq ray_{\alpha} \leq lens_{max}$$

Deux issues sont alors possible :

- la longueur d'onde entre dans l'intervalle et le rayon est renouvelé après la lentille
- ou elle ne rentre pas dans l'intervalle défini et le rayon sera terminé par la lentille

et ce même si le rayon tiré est tangent à la lentille. En effet, si la droite représentant le rayon est tangent à l'ellipse représentant la lentille c'est qu'un point d'intersection existe entre les deux. Il est donc de ce point d'agir comme l'aurait fait l'entière lentille à tout autre point même si cette réaction particulière peut être discutable.

3.2.5 mirror.hpp

Cette classe est, par héritage, un élément ainsi qu'une droite. Le miroir peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communiquer avec le niveau auquel il est lié. Le miroir est, plus précisément, un segment de droite délimité par deux points. Puisque le miroir effectue une rotation autour d'un point $p \in \text{miroir}$, il n'est pas directement délimité par ses deux extrémités mais bien par sa longueur, son angle, la

position absolue (cartésienne) et relative (distance par rapport à une extrémité) de son point de rotation qui définissent, dynamiquement, ses points délimiteurs. Pour trouver les deux points il suffit de :

1. extrémité gauche = $(x_{pivot} - \text{position absolue}, y_{pivot})$
2. extrémité droite = $(x_{gauche} + \text{taille du segment}, y_{pivot})$
3. rotation des deux extrémité autour du point de pivot.

3.2.6 nuke.hpp

Cette classe est, par héritage, un élément ainsi qu'une ellipse. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La bombe est une ellipse particulière puisqu'elle possède un $x_{radius} = y_{radius}$ lui donnant comme caractéristique d'être un cercle. Cet objet circulaire est l'objet à éviter lors d'une partie puisqu'il amène directement à la fin d'une partie.

3.2.7 ray.hpp

Cette classe est, par héritage, une droite. Ce rayon représente l'ensemble des points parcouru par un rayon laser et subit les interactions de son environnement comme dans la vie réelle :

- il sera reflété par un miroir selon le principe de réflexion

$$\alpha_i = \alpha_r$$

- il sera arrêté par les objets opaques (murs, source, destination).

Le rayon déclenche les réactions des éléments du jeu via la méthode "reactToRay" des héritiers d'Element.

3.2.8 source.hpp

Cette classe est, par héritage, un élément ainsi qu'un rectangle. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. La source est le point d'émission du tout premier rayon du jeu selon un sens et une direction définis par l'angle d'émission, qui lui est propre, depuis un point donné.

3.2.9 wall.hpp

Cette classe est, par héritage, un élément ainsi qu'une droite. Il peut donc être modéliser graphiquement, entant que cette dernière, dans le plan et communique avec le niveau auquel il est lié. Le mur est défini par deux points qui en font un segment de droite. Il est un objet fixe du jeu qui arrête un rayon et lui défini un point qui le transforme en segment de droite.

3.2.10 level.hpp

3.2.11 levelfactory.hpp

Le créateur de niveau est un “namespace” de méthodes qui va s’occuper de lire les données des éléments dans un fichier. Pour ce faire, il s’occupera de lire chaque type d’objet selon des règles différentes :

```
levelFactory :: getSource ;  
levelFactory :: getDestination ;  
levelFactory :: getCrystal ;  
levelFactory :: getLens ;  
levelFactory :: getNuke ;  
levelFactory :: getWall ;  
levelFactory :: getMirror ;
```

3.3 exception

3.3.1 starlightexception.hpp

Il est nécessaire, pour bon nombre des classes créées, de valider les arguments passés en paramètre dans le but de ne pas produire d’objets incohérents par rapport à l’analyse préalable du travail à fournir. Pour ce faire, des exceptions doivent être levées quand une instantiation créera un objet non désiré. Cette classe hérite de `std : exception` appartenant à la librairie standard. Elle n’a aucune capacité supplémentaire mise à part être spécifique à ce projet.

Les différentes classes pouvant lever cette exception sont :

crystal si la taille de son rayon ne lui permet pas d’exister dans le plan,
lens si son intervalle de longueur d’onde n’est pas cohérent,
level si ses dimensions ne lui permettent pas d’exister dans le plan,
mirror si ses dimensions ne lui permettent pas d’exister dans le plan, si sa position ou son angle n’entre pas dans les limites imposées,
nuke si la taille de son rayon ne lui permet pas d’exister dans le plan,
ray si sa longueur d’onde n’entre pas dans l’intervalle cohérent imposé,
source si sa longueur d’onde n’entre pas dans l’intervalle cohérent imposé,
wall si ses points déterminants ne lui permettent pas d’exister dans le plan,
ellipse si ses dimensions ne lui permettent pas d’exister dans le plan,
rectangle si ses dimensions ne lui permettent pas d’exister dans le plan.

3.4 view

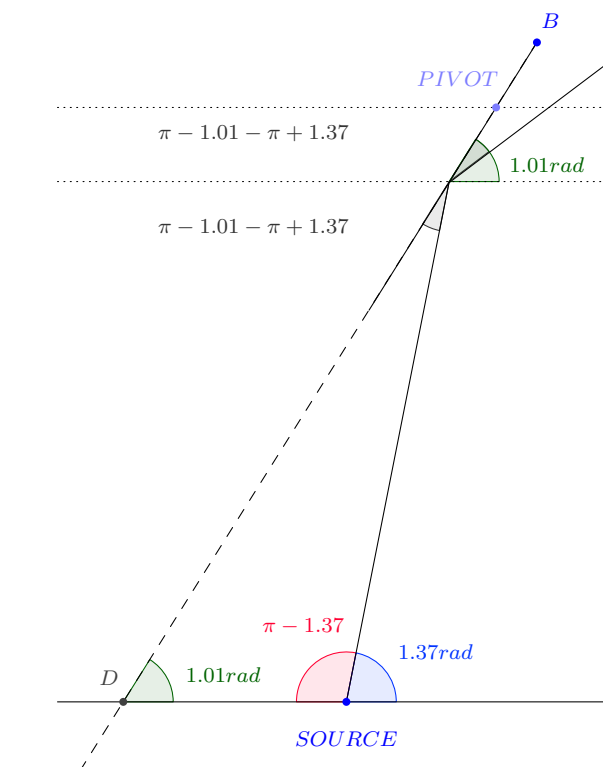
Chapitre 4

Structure générale du programme

Chapitre 5

Détail des algorithmes utilisés

5.1 Algorithme de réflexion



α l'angle du nouveau rayon

β l'angle du miroir

γ l'angle incident

ϵ l'angle de la source

$\gamma = (\pi - \beta - (\pi - \epsilon))$ par la somme des angles intérieurs d'un triangle

valant πrad ,
 $\alpha = \beta - \gamma$ par les angles correspondants

5.2 Algorithme d'intersection

5.2.1 Intersection de deux droites

Soient

$$\begin{cases} y = m_1 \cdot x + p_1 \\ y = m_2 \cdot x + p_2 \end{cases}$$

deux droites sécantes ($m_1 \neq m_2$), résoudre l'équation

$$m_1 \cdot x + p_1 = m_2 \cdot x + p_2$$

$$m_1 \cdot x - m_2 \cdot x = p_2 - p_1$$

$$x \cdot (m_1 - m_2) = p_2 - p_1$$

$$x = \frac{(p_2 - p_1)}{(m_1 - m_2)}$$

nous donne l'abscisse du point d'intersection des deux droites. Il suffit ensuite de trouver y à l'aide d'une des deux équations.

5.2.2 Intersection d'une droite et d'un rectangle

Pour trouver les intersections entre un rectangle et une droite, il suffit de chercher les intersections entre une droite et les quatre droites qui composent le rectangle (Cf. "Intersection de deux droites").

5.2.3 Intersection d'une droite et d'une ellipse

Soient

$$\begin{cases} y = m \cdot x + p \\ \frac{(x-c_x)^2}{x_{radius}^2} + \frac{(y-c_y)^2}{y_{radius}^2} = 1 \end{cases}$$

une droite et une ellipse quelconque, substituons la droite à l'équation de l'ellipse

$$\begin{cases} y = m \cdot x + p \\ \frac{(x-c_x)^2}{x_{radius}^2} + \frac{((m \cdot x + p) - c_y)^2}{y_{radius}^2} = 1 \end{cases}$$

$$\begin{cases} y = m \cdot x + p \\ \frac{(x-c_x)^2 * y_{radius}^2}{x_{radius}^2 * y_{radius}^2} + \frac{((m \cdot x + p) - c_y)^2 * x_{radius}^2}{y_{radius}^2 * x_{radius}^2} = 1 \end{cases}$$

Chapitre 6

Test effectués

6.1 Framework de test

Le Framework de test utilisé est “Catch”¹. Ses avantages sont :

- un simple header est requis,
- **TEST_CASE** créant un bloc de tests,
- **SECTION** créant un sous bloc de tests,
- un ensemble de macros d’assertions sont disponibles :
 - **REQUIRE** qui attend une expression vraie,
 - **REQUIRE_FALSE** qui attend une expression fausse,
 - **REQUIRE_THROWS_AS** qui attend une erreur de type défini,
 - **REQUIRE_NO_THROW** qui n’attend pas d’erreur.

6.2 Tests unitaire

Chaque classe a été soumise à une batterie de tests unitaires. Pour se faire, chacune de ses méthodes s’est vu confirmé son bon fonctionnement au travers de cas dit “limites” et de cas dit “standards”.

Les fichiers sources concernés se situent dans le dossier `test/` et peuvent être effectués sous certaines conditions².

1. <https://github.com/philsquared/Catch>

2. en effet, il faudra au préalable rendre la méthode `main` de l’application inutilisable pour permettre aux tests de s’appliquer exclusivement

Chapitre 7

Conclusion

Annexe A

Références